[2022 Operating System] Project Abstract

Group members: 407410003 常亦德、407410065 王宸潤、408410042 林靖紳

Abstract

Parallel or multiple threads and processes are often come into the problem of synchronization. In “Topic 6 Synchronization”, we will learn what is race condition, critical sections problem and its solutions: Mutex and Semaphore.
In Linux kernel, the strategies of lock can mainly devided into two part: spinning and blocking (sleeping) lock.
In the slides of lecture 6, we will learn the blocking strategy, which make thread sleep until the lock is free and then put it back into run queue. Sleeping locks are suitable for locks that are held in a longer periods; however, the overhead of context switching and dispatching as well as the consequent increases in cache miss.
In this project, we will introduce another lock: Spinlock. We will implement the spinlocks, analysis its efficacy and compare it to Mutex and Sephamore in our project. Furthermore, we will also analyze the advantage and disadvantage of spinlocks and try to make some improvements on spinlock.

Introduction

As the number of multi-core systems increases, more processes or threads execute simultaneously on different cores. When multiple processes execute the same code or access same memory or any shared variable in this situation, the output or the value of the shared variables will probably be wrong; however, all the processes doing the race return that their output value are correct. This condition is known as a race condition.
A race condition may occur inside a critical section, where only one process is allowed to access the code segment at a time, when the result of multiple threads execution differs depending on the executing order.
The critical section contains shared variables that need to

be sychronized to maintain the consistency of data variables. Therefore, critical section problem means that we should designing a way to access shared resources for cooperative processes without creating data inconsistencies.
Most of shared memory multiprocessor architectures provide hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. Take RISC-V for example, it provide RV32A Instruction set which is a set of atomic operations.
These atomic operations are used to manipulate locks; when a processor is accessing a data structure, the lock is busy, so other processors requesting access must wait, also calles as “busy waiting.”
For small critical sections, spinning for a lock to be released is more efficient than relinquishing the processor to do other works; however, its overhead is spin-waiting will slow down other processors by consuming communication bandwidth.
In this project we will look into these problems, detaily develop how it happen and try to optimize the drawbacks. Therefore, we will devided our works into three part:

- Implementation
  - Implement spinlock in C language
  - Design a testing structure for analyzing its efficacy
- Analysis
  - Record and aggregate the data and compare the data with other locks
  - Analyze pros and cons of Spinlock, and discuss under which circumstances would lead to the outcome
- Improvement
  - Based on the results of our analysis, try to make some improvements to our Spinlocks code
  - Try to optimize the overhead of cores communication bandwidth